# Likelion Study #AI. Team2

**Part. CNN**
**2022-01-03**

01

# Part 1

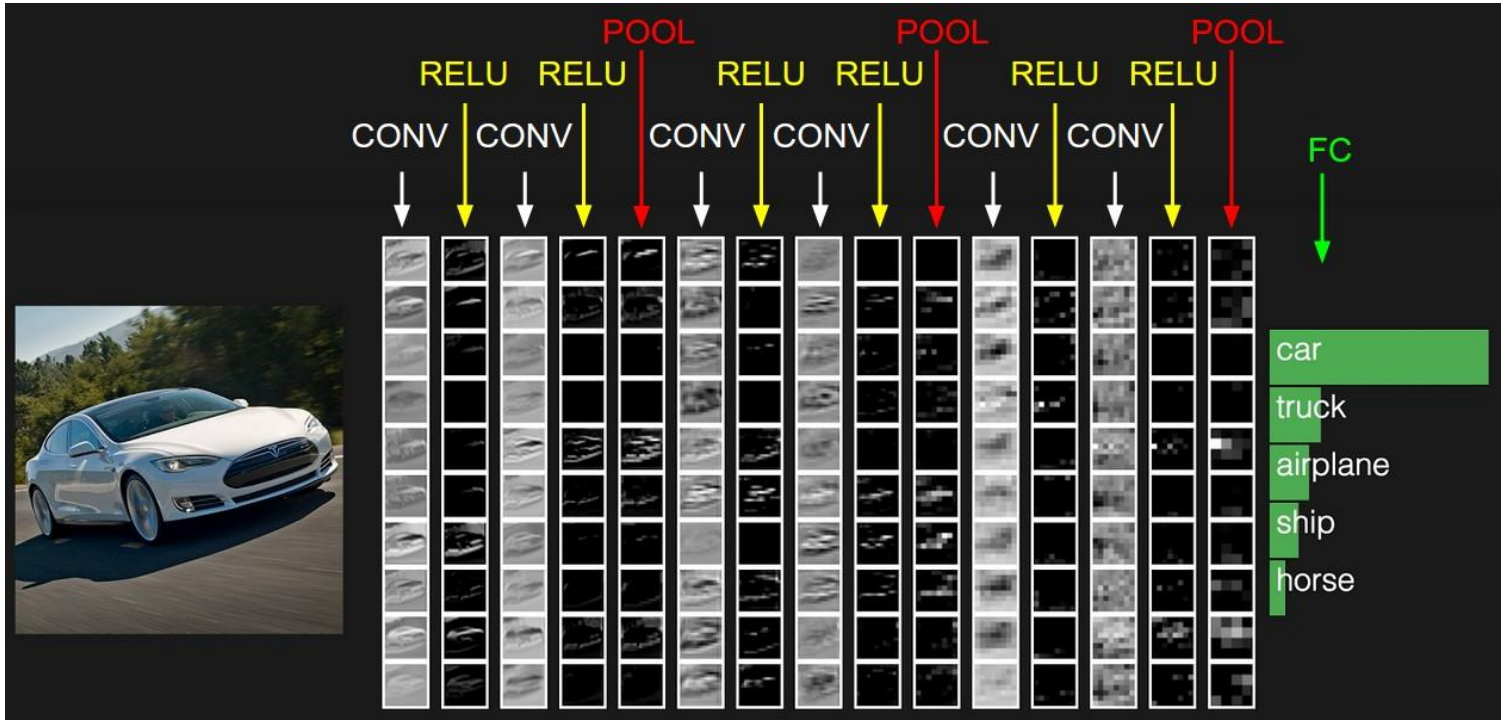
1. 합성곱 신경망
2. 합성곱 연산
3. 풀링

# Convolutional Neural Network
# 합성곱 신경망

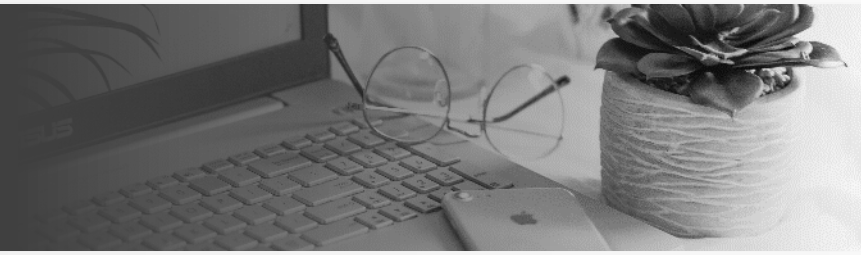
합성곱 신경망 (CNNs, ConvNets)

합성곱 신경망 (Convolutional Neural Network)는 이미지 처리에 탁월한 성능을 보이는 신경망.

합성곱 신경망은 합성곱층 (Convolution layer)과 풀링층 (Pooling layer)으로 구성됨,

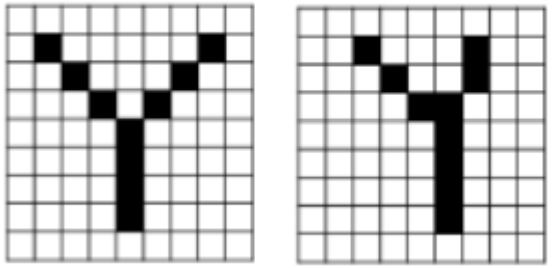


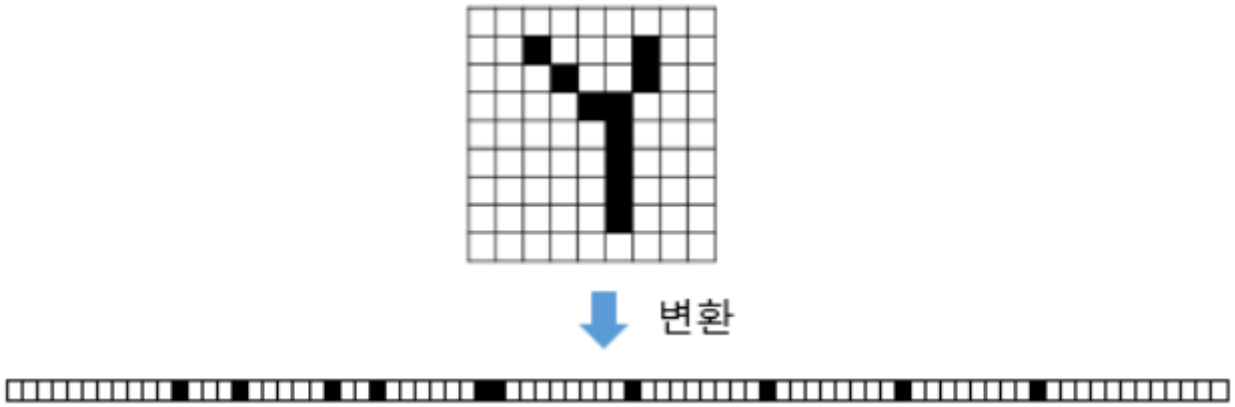
# CNN의 대두



이및 처리를 하기 위해서 앞서 배운 다층 퍼셉트론을 사용하게 되면 한계 존재.

→ 공간적 구조(spatial structure) 정보가 유실됨.

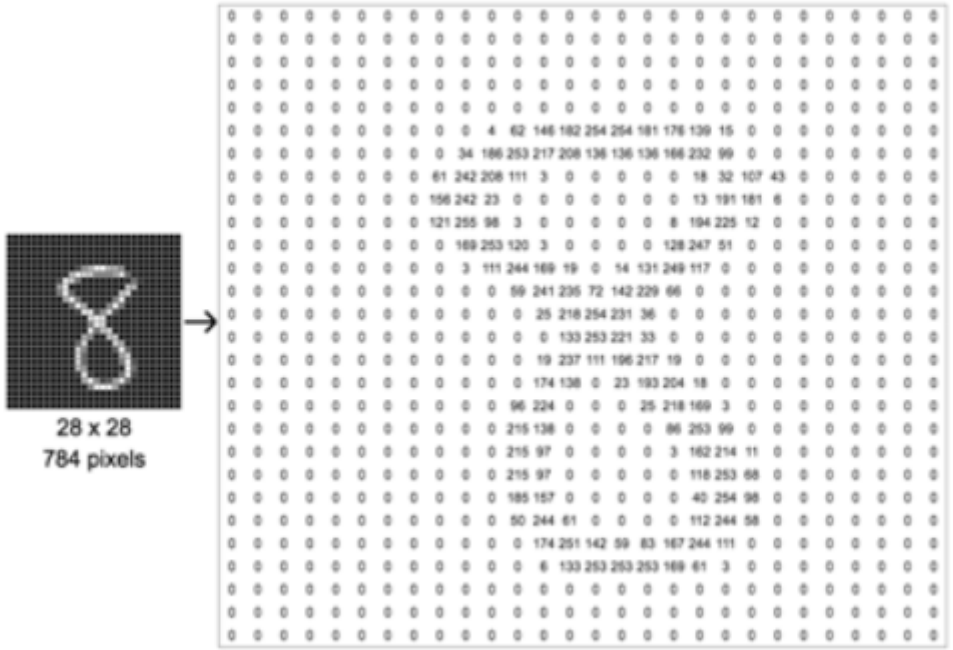


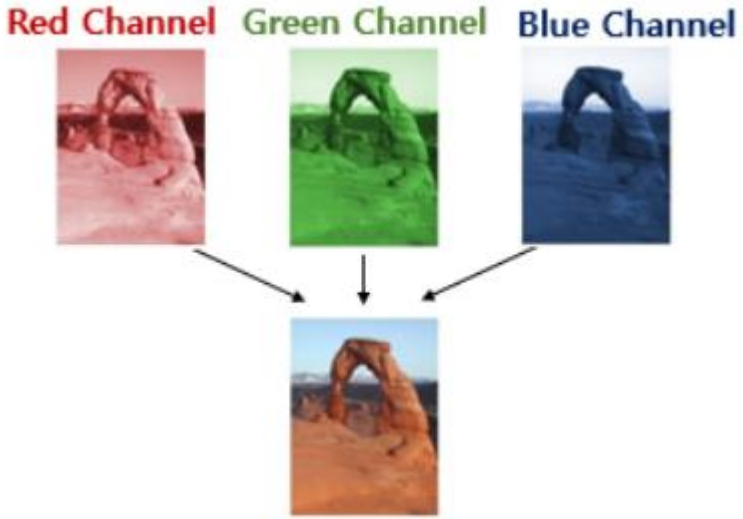


# Channel
# 채널

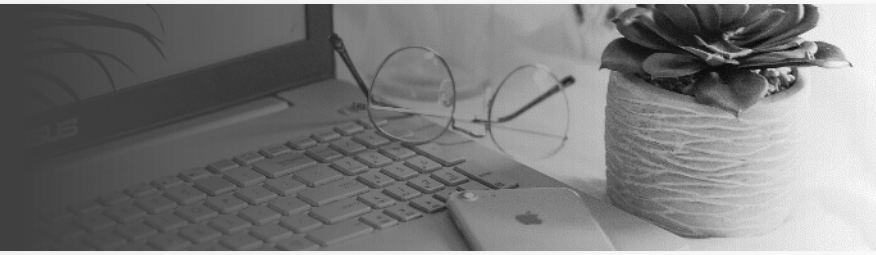
이미지는 (높이, 너비, 채널)이라는 Tensor

흑백 이미지는 채널 1, 컬러 이미지 채널 3 (RGB)



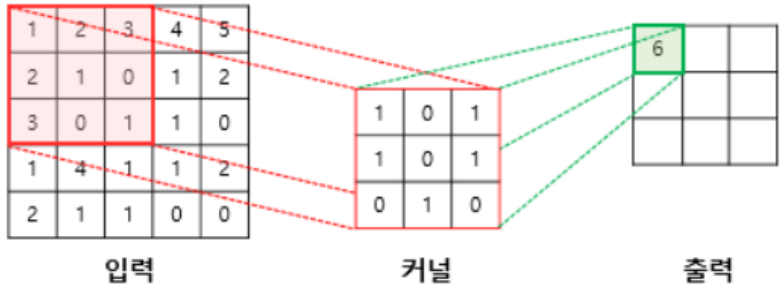



# Convolution
# 합성곱

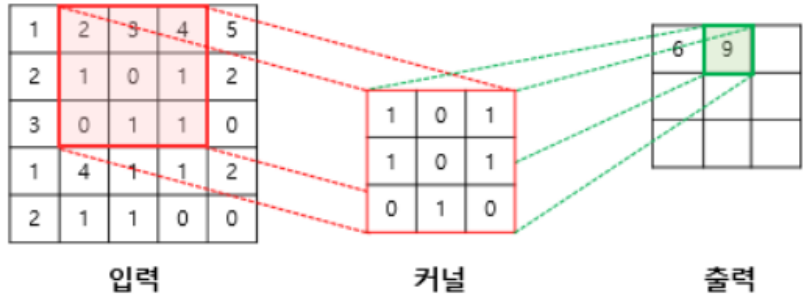


합성곱층은 합성곱 연산을 총해서 이미지의 특징을 추출.

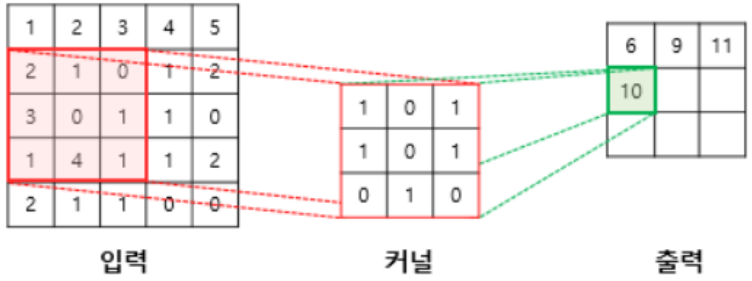
커널(Kernel)이라는 n x m 크기의 행렬로 이미지를 처음부터 끝까지 훑으면서 n x m크기의 겹치는 부분의 각 이미지와 커널의 원소의 값을 곱해서 모두 더한 값을 출력




(1×1) + (2×0) + (3×1) + (2×1) + (1×0) + (0×1) + (3×0) + (0×1) + (1×0) = 6




(2×1) + (3×0) + (4×1) + (1×1) + (0×0) + (1×1) + (0×0) + (1×1) + (1×0) = 9




(2×1) + (1×0) + (0×1) + (3×1) + (0×0) + (1×1) + (1×0) + (4×1) + (1×0) = 10

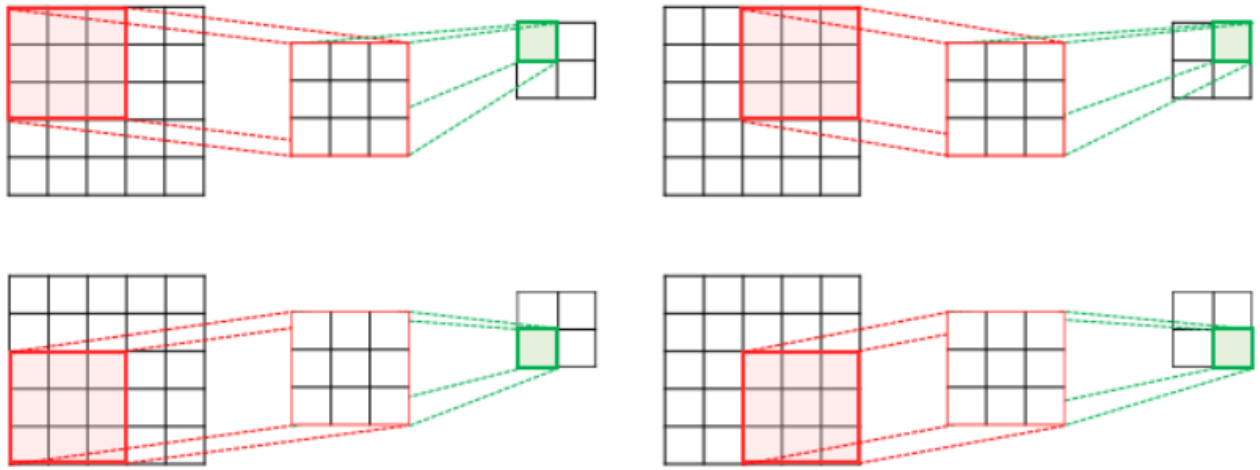
# Convolution
# 합성곱 (Cont.)

| | | |
|---|---|---|
| 6 | 9 | 11 |
| 10 | 4 | 4 |
| 7 | 7 | 4 |

**특성 맵(feature map)**

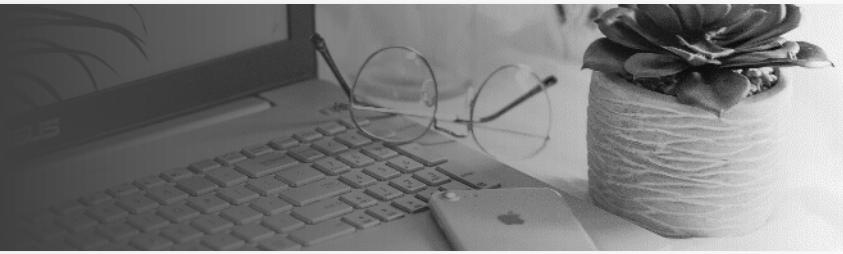
입력으로부터 커널을 사용하여 합성곱 연산을 통해 나온 결과는 특성 맵 (feature map)이라고 함.

이동 범위에 대해서 사용자가 직접 정할 수 있으며 전 예제에서는 이본 범위가 한 칸이고 이동 범위를 **스트라이드 (stride)**라고 함.



Stride 2

# Padding
# 패딩



합성곱 연산을 통해 얻어진 특성 맵은 입력보다 크기가 작아지는 특징이 있음.

합성곱 연산 후에도 특성 맵의 크기가 입력의 크기와 동일하게 유지하고 싶다면 padding을 사용.

주로 테두리의 값을 0으로 채우는 제로 패딩 (zero padding)을 사용.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 1 | 0 | 1 | 2 |
| 3 | 0 | 1 | 1 | 0 |
| 1 | 4 | 1 | 1 | 2 |
| 2 | 1 | 1 | 0 | 0 |

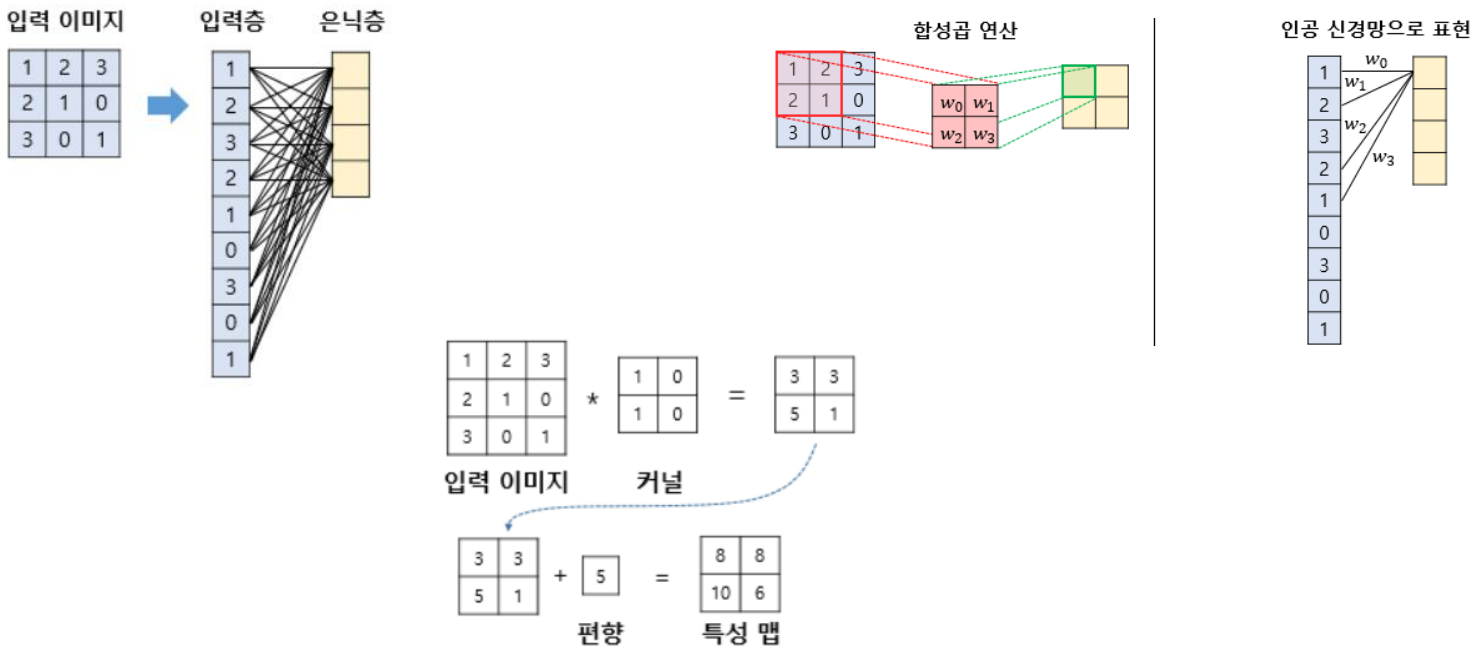
패딩 전

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 0 |
| 0 | 2 | 1 | 0 | 1 | 2 | 0 |
| 0 | 3 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 4 | 1 | 1 | 2 | 0 |
| 0 | 2 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

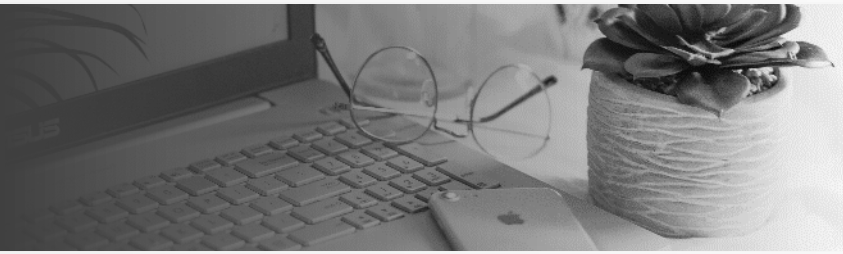
패딩 후

# 가중치화
# 편향

다층 퍼셉트론보다 합성곱 신경망은 적은 수의 가중치를 사용하여 공간적 구조 정보를 보존한다는 특징.



# 특성 맵의 크기 계산

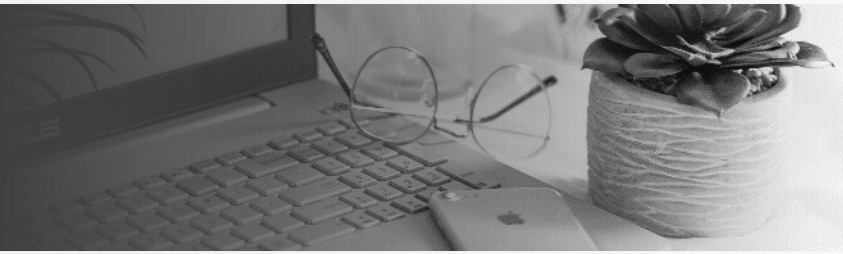


- $I_h$ : 입력의 높이
- $I_w$ : 입력의 너비
- $K_h$ : 커널의 높이
- $K_w$ : 커널의 너비
- $S$ : 스트라이드
- $O_h$ : 특성 맵의 높이
- $O_w$ : 특성 맵의 너비

$$O_h = floor(\frac{I_h - K_h + 2P}{S} + 1)$$

$$O_w = floor(\frac{I_w - K_w + 2P}{S} + 1)$$

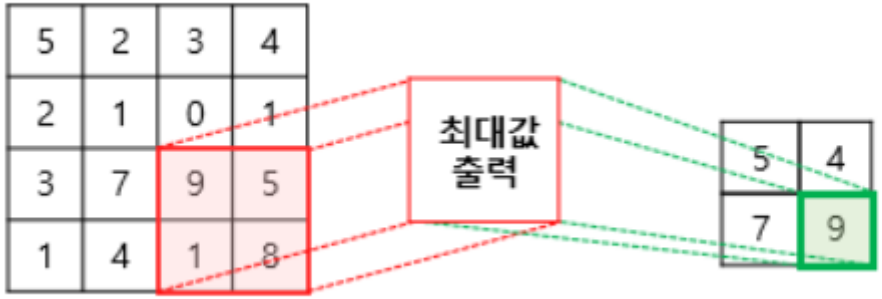
# Pooling
# 풀링



일반적으로 합성층(합성곱 연산 + 활성화 함수) 다음에는 풀링 층을 추가하는 것이 일반적.

풀링 층에서는 특성 맵을 다운샘플링하여 특성 맵의 크기를 줄이는 풀링 연산.

일반적으로 최대 풀링과 평균 풀링을 사용.

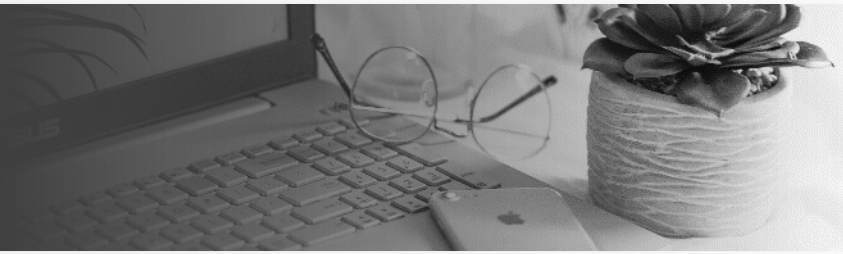


02

# Part 2

## CNN으로 MNIST 분류하기

# 1. 모델 이해하기

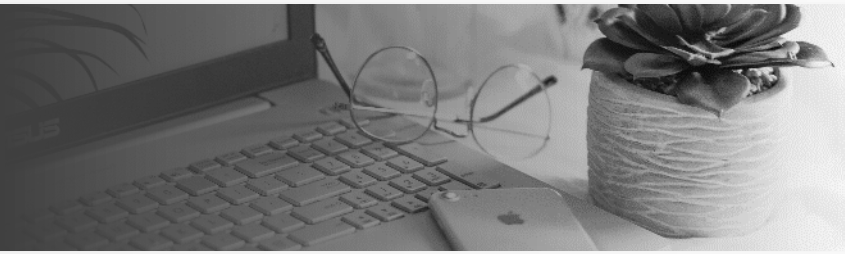


## 모델의 구조

```
# 1번 레이어 : 합성곱층(Convolutional layer)
합성곱(in_channel = 1, out_channel = 32, kernel_size=3, stride=1, padding=1) + 활성화 함수 ReLU
맥스풀링(kernel_size=2, stride=2))

# 2번 레이어 : 합성곱층(Convolutional layer)
합성곱(in_channel = 32, out_channel = 64, kernel_size=3, stride=1, padding=1) + 활성화 함수 ReLU
맥스풀링(kernel_size=2, stride=2))

# 3번 레이어 : 전결합층(Fully-Connected layer)
특성맵을 펼친다. # batch_size × 7 × 7 × 64 → batch_size × 3136
전결합층(뉴런 10개) + 활성화 함수 Softmax
```

2개의 합성곱층과 1개의 전결합층으로 구성 합성곱층에는
합성곱(nn.Conv2d) + 활성화 함수(nn.ReLU) + 맥스풀링(nn.MaxPoold2d) 포함

# 2. 모델 구현하기



## 1. 준비

```
import torch
import torch.nn as nn
```

```
# 배치 크기 × 채널 × 높이(height) × 너비(widht)의 크기의 텐서를 선언
inputs = torch.Tensor(1, 1, 28, 28)
print('텐서의 크기 : {}'.format(inputs.shape))
```

```
텐서의 크기 : torch.Size([1, 1, 28, 28])
```

# 2. 모델 구현하기

## 2. 합성곱층과 풀링 선언하기

```
conv1 = nn.Conv2d(1, 32, 3, padding=1)
print(conv1)
```

Conv2d(입력채널 수, 출력채널 수, 커널사이즈, … )

커널 = 필터

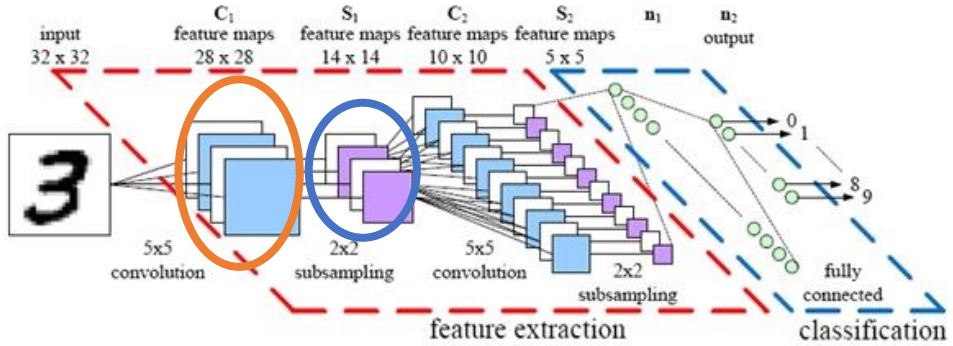
```
Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

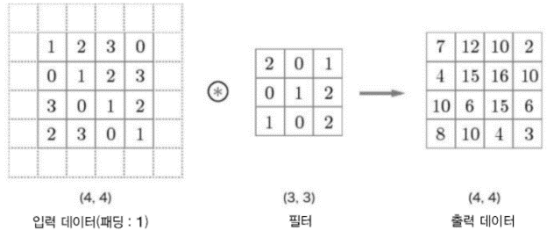
```
conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
print(conv2)
```

```
Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

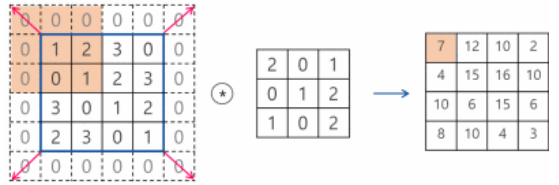
```
pool = nn.MaxPool2d(2)
print(pool)
```

```
MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fa
```

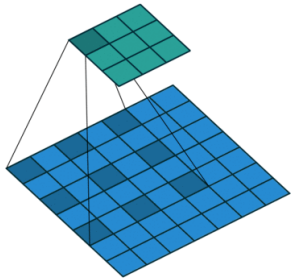



Padding : 필터를 적용 했을 때 가장자리가 사라지면서 사이즈가 줄어드는 것을 방지



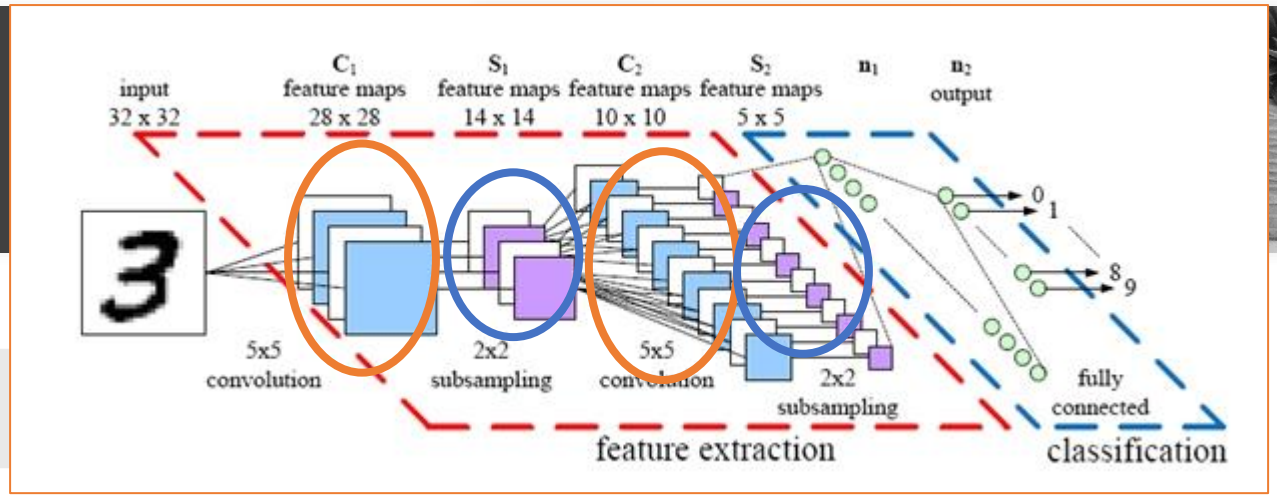
Stride : 필터를 어느정도 간격으로 적용시킬 것인가?



Dilation

Ceil_mode
- 바닥함수
- 천장함수

# 2. 모델 구현하기




## 3. 구현체를 연결하여 모델 만들기

```
out = conv1(inputs)
print(out.shape)
```

```
torch.Size([1, 32, 28, 28])
```
채널 32개, 넓이 & 높이 28 ( 넓이와 높이는 Padding으로 크기 보존 )

```
out = pool(out)
print(out.shape)
```

```
torch.Size([1, 32, 14, 14])
```
Pooling으로 중요한 요소만 담아서 압축

```
out = conv2(out)
print(out.shape)
```

```
torch.Size([1, 64, 14, 14])
```
채널 64개, 넓이 & 높이 14 ( 넓이와 높이는 Padding으로 크기 보존 )

```
out = pool(out)
print(out.shape)
```

```
torch.Size([1, 64, 7, 7])
```
Pooling으로 중요한 요소만 담아서 압축

필터 적용 후

# 2. 모델 구현하기

## 3. 구현체를 연결하여 모델 만들기

```
# 첫번째 차원인 배치 차원은 그대로 두고 나머지는 펼쳐라
out = out.view(out.size(0), -1)
print(out.shape)
```

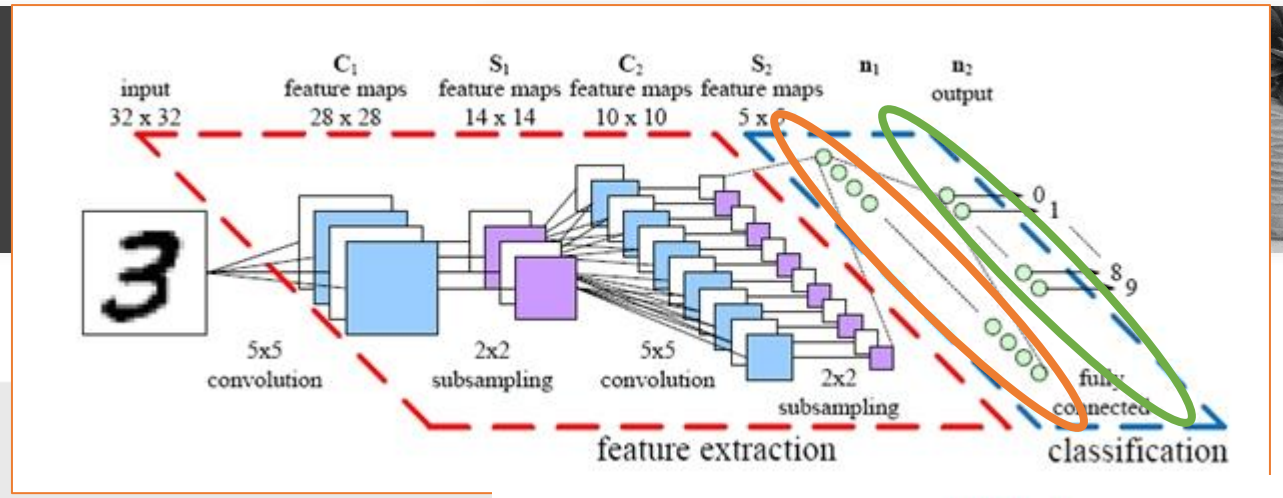
1차원으로 모두 나열

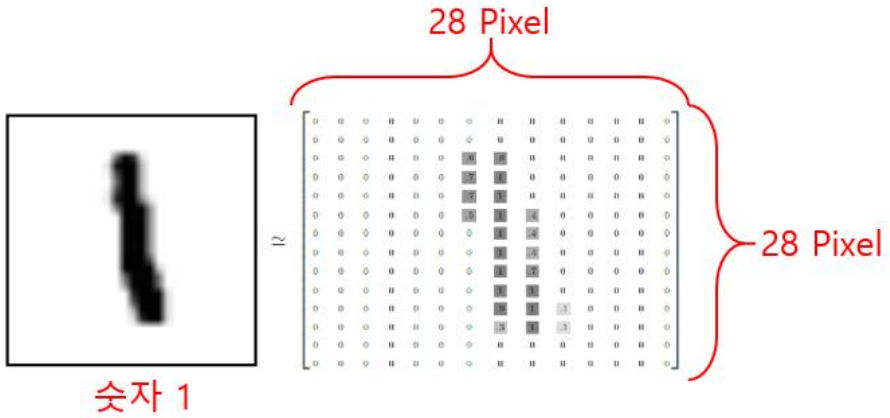
```
torch.Size([1, 3136])
```

```
fc = nn.Linear(3136, 10) # input_dim = 3,136, output_dim = 10
out = fc(out)
print(out.shape)
```

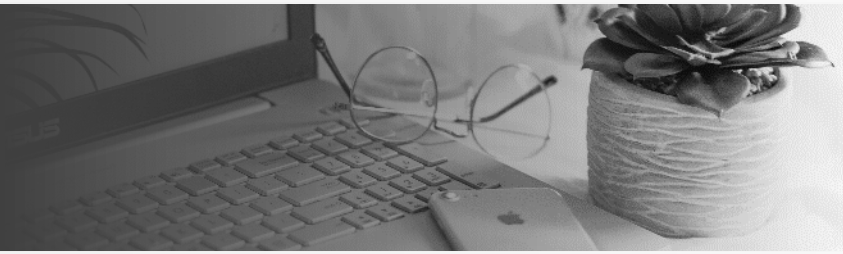
Fully Connected Layer

```
torch.Size([1, 10])
```



# 3. CNN으로 MNIST 분류하기



```
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.nn.init
```

필요한 모듈 Import

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 랜덤 시드 고정
torch.manual_seed(777)

# GPU 사용 가능일 경우 랜덤 시드 고정
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
```

계산방법 선택

```
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

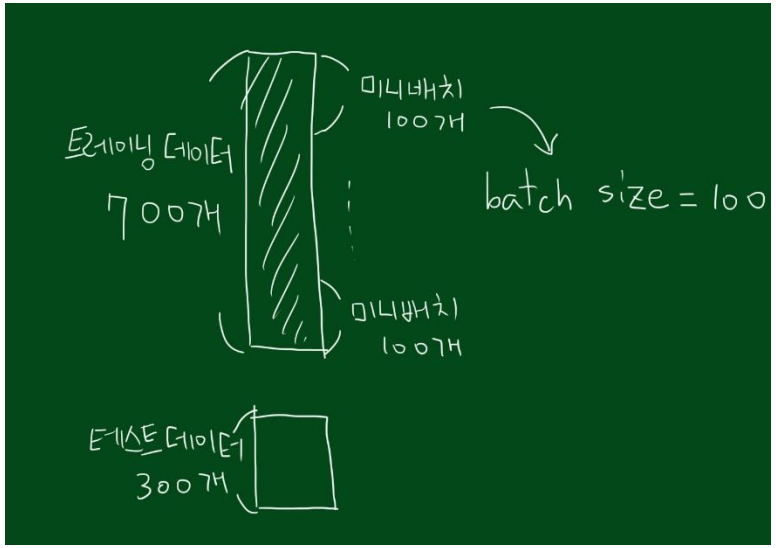
파라미터 설정

```
mnist_train = dsets.MNIST(root='MNIST_data/', # 다운로드 경로 지정
                          train=True, # True를 지정하면 훈련 데이터로 다운로드
                          transform=transforms.ToTensor(), # 텐서로 변환
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/', # 다운로드 경로 지정
                         train=False, # False를 지정하면 테스트 데이터로 다운로드
                         transform=transforms.ToTensor(), # 텐서로 변환
                         download=True)
```

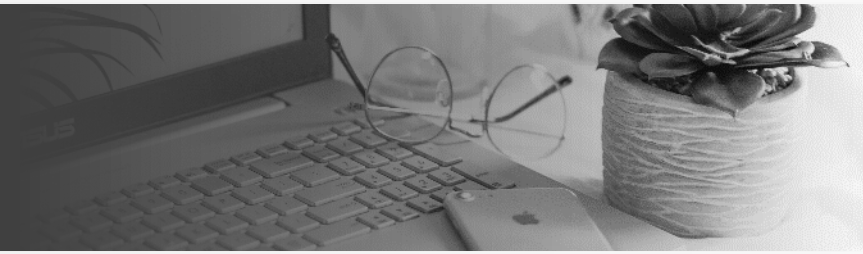
MNIST 데이터 로드

```
data_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True)
```

미니배치 데이터 세팅



# 3. CNN으로 MNIST 분류하기



모델 설계

```
class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        # 첫번째층
        # ImgIn shape=(?, 28, 28, 1)
        #    Conv     -> (?, 28, 28, 32)
        #    Pool     -> (?, 14, 14, 32)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))

        # 두번째층
        # ImgIn shape=(?, 14, 14, 32)
        #    Conv      ->(?, 14, 14, 64)
        #    Pool      ->(?, 7, 7, 64)
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))

        # 전결합층 7x7x64 inputs -> 10 outputs
        self.fc = torch.nn.Linear(7 * 7 * 64, 10, bias=True)

        # 전결합층 한정으로 가중치 초기화
        torch.nn.init.xavier_uniform_(self.fc.weight)
```

1번 레이어

2번 레이어

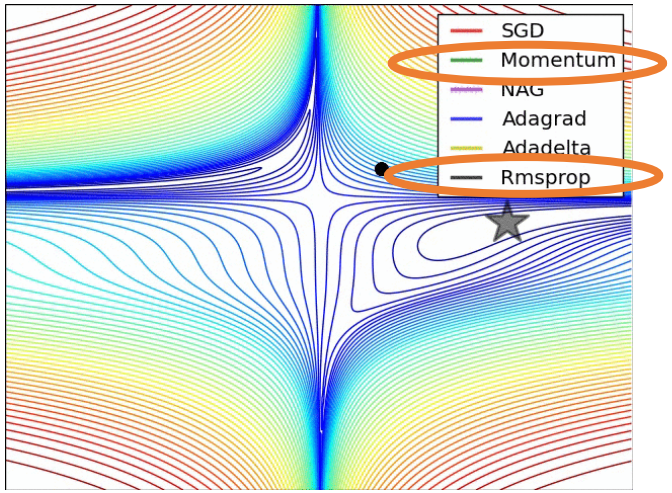
Fully Connected Layer

```
    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)   # 전결합층을 위해서 Flatten
        out = self.fc(out)
        return out
```

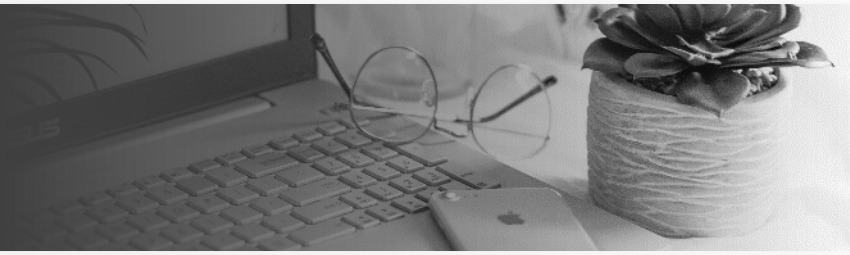
```
# CNN 모델 정의
model = CNN().to(device)
```

Device = CPU or GPU

```
criterion = torch.nn.CrossEntropyLoss().to(device)    # 비용 함수에 소프트맥스 함수 포함되어져 있음.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```



# 3. CNN으로 MNIST 분류하기



```
total_batch = len(data_loader)
print('총 배치의 수 : {}'.format(total_batch))
```

```
총 배치의 수 : 600
```

```
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

훈련데이터 60000개

```
for epoch in range(training_epochs):
    avg_cost = 0

    for X, Y in data_loader:


        X = X.to(device)
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = model(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('[Epoch: {:>4}] cost = {:>.9}'.format(epoch + 1, avg_cost))
```

최적의 모델 탐색

```
with torch.no_grad():
    X_test = mnist_test.test_data.view(len(mnist_test), 1, 28, 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = model(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())
```

테스트 데이터 세팅

최적의 모델 탐색

```
Accuracy: 0.9883000254631042
```

**x, y, z**
**(3, 4, 3)**

x = 0

| y = 0 | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 2 | 1 | 4 |
| 2 | 5 | 2 | 1 |
| 3 | 6 | 3 | 2 |

x = 1

| y = 0 | 5 | 1 | 3 |
|---|---|---|---|
| 1 | 1 | 3 | 4 |
| 2 | 4 | 2 | 6 |
| 3 | 3 | 9 | 3 |

x = 2

| y = 0 | 4 | 5 | 6 |
|---|---|---|---|
| 1 | 7 | 4 | 3 |
| 2 | 2 | 1 | 5 |
| 3 | 4 | 3 | 1 |

**np.argmax(a, axis = 1)**

axis가 1일 경우 y 축 기준

비교 값 모음

| | | |
|---|---|---|
| 1,2,5,6 | 2,1,2,3 | 3,4,1,2 |
| 5,1,4,3 | 1,3,2,9 | 3,4,6,3 |
| 4,7,2,4 | 5,4,1,3 | 6,3,5,1 |

결과 값

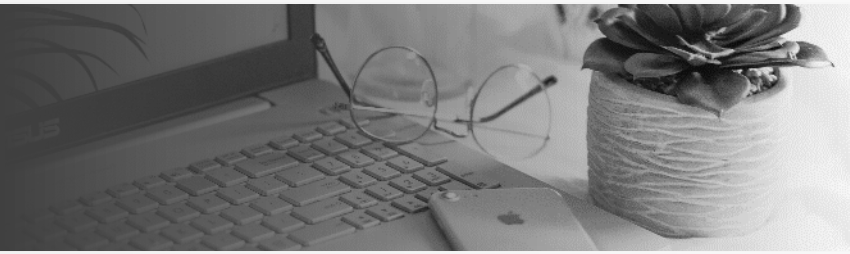
| | | |
|---|---|---|
| 3 | 3 | 1 |
| 0 | 3 | 2 |
| 1 | 0 | 0 |

03

# Part 3

## 깊은 CNN으로 MNIST 분류하기

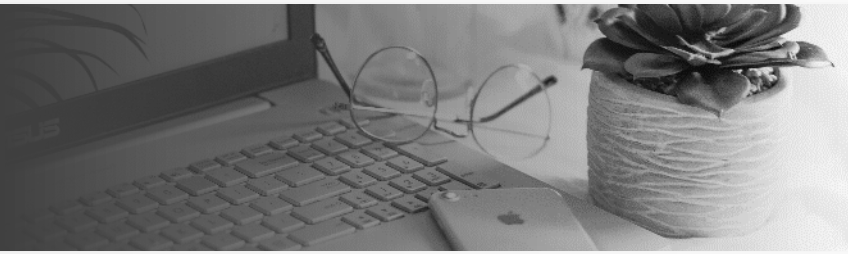
# 1. 모델 이해하기



```
# 1번 레이어 : 합성곱층(Convolutional layer)
합성곱(in_channel = 1, out_channel = 32, kernel_size=3, stride=1, padding=1) + 활성화 함수 ReLU
맥스풀링(kernel_size=2, stride=2))

# 2번 레이어 : 합성곱층(Convolutional layer)
합성곱(in_channel = 32, out_channel = 64, kernel_size=3, stride=1, padding=1) + 활성화 함수 ReLU
맥스풀링(kernel_size=2, stride=2))

# 3번 레이어 : 합성곱층(Convolutional layer)
합성곱(in_channel = 64, out_channel = 128, kernel_size=3, stride=1, padding=1) + 활성화 함수 ReLU
맥스풀링(kernel_size=2, stride=2, padding=1))

# 4번 레이어 : 전결합층(Fully-Connected layer)
특성맵을 펼친다. # batch_size × 4 × 4 × 128 → batch_size × 2048
전결합층(뉴런 625개) + 활성화 함수 ReLU

# 5번 레이어 : 전결합층(Fully-Connected layer)
전결합층(뉴런 10개) + 활성화 함수 Softmax
```

이전과 동일

# 2. 깊은 CNN으로 MNIST 분류하기



```
import torch
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.nn.init
```

# 필요한 도구 import

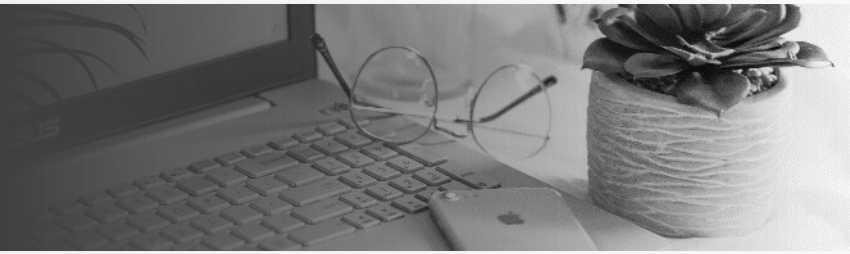
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# 랜덤 시드 고정
torch.manual_seed(777)

# GPU 사용 가능일 경우 랜덤 시드 고정
if device == 'cuda':
    torch.cuda.manual_seed_all(777)
```

# GPU 사용 가능하면 'cuda', 안되면 'cpu'

```
learning_rate = 0.001
training_epochs = 15
batch_size = 100
```

# 파라미터 설정

# 2. 깊은 CNN으로 MNIST 분류하기



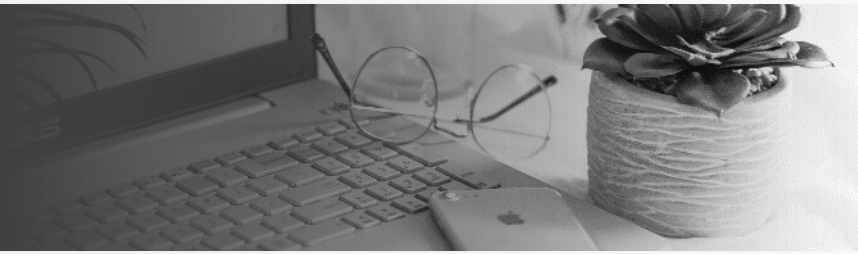
```
mnist_train = dsets.MNIST(root='MNIST_data/', # 다운로드 경로 지정
                          train=True, # True를 지정하면 훈련 데이터로 다운로드
                          transform=transforms.ToTensor(), # 텐서로 변환
                          download=True)

mnist_test = dsets.MNIST(root='MNIST_data/', # 다운로드 경로 지정
                         train=False, # False를 지정하면 테스트 데이터로 다운로드
                         transform=transforms.ToTensor(), # 텐서로 변환
                         download=True)
```

# 데이터셋 정의

```
data_loader = torch.utils.data.DataLoader(dataset=mnist_train,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          drop_last=True)
```

# 배치크기 지정

# 2. 깊은 CNN으로 MNIST 분류하기



```
class CNN(torch.nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.keep_prob = 0.5
        # L1 ImgIn shape=(?, 28, 28, 1)
        #    Conv     -> (?, 28, 28, 32)
        #    Pool     -> (?, 14, 14, 32)
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))
        # L2 ImgIn shape=(?, 14, 14, 32)
        #    Conv      ->(?, 14, 14, 64)
        #    Pool      ->(?, 7, 7, 64)
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2))
```

# 모델 설계

$$O_h = floor(\frac{I_h - K_h + 2P}{S} + 1)$$

$$O_w = floor(\frac{I_w - K_w + 2P}{S} + 1)$$

**Layer 1 :**

(?, 28, 28, 1)

⋮ Conv(합성곱연산)

너비: $\frac{28-3+2}{1} + 1 = 28$

높이: $\frac{28-3+2}{1} + 1 = 28$

(?, 28, 28, 32)

⋮ Pool(맥스풀링)

너비: $\frac{28-2}{2} + 1 = 14$

높이: $\frac{28-2}{2} + 1 = 14$

(?, 14, 14, 32)

**Layer 2 :**

(?, 14, 14, 32)

⋮ Conv(합성곱연산)

너비: $\frac{14-3+2}{1} + 1 = 14$

높이: $\frac{14-3+2}{1} + 1 = 14$

(?, 14, 14, 64)

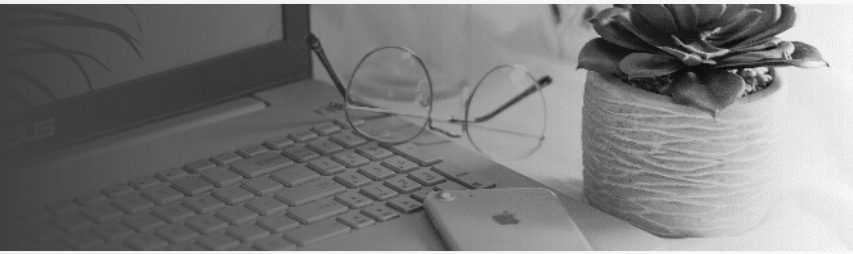
⋮ Pool(맥스풀링)

너비: $\frac{14-2}{2} + 1 = 7$

높이: $\frac{14-2}{2} + 1 = 7$

(?, 7, 7, 64)

# 2. 깊은 CNN으로 MNIST 분류하기



```
# L3 ImgIn shape=(?, 7, 7, 64)
#    Conv      ->(?, 7, 7, 128)
#    Pool      ->(?, 4, 4, 128)
self.layer3 = torch.nn.Sequential(
    torch.nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=1))

# L4 FC 4x4x128 inputs -> 625 outputs
self.fc1 = torch.nn.Linear(4 * 4 * 128, 625, bias=True)
torch.nn.init.xavier_uniform_(self.fc1.weight)     # 가중치 초기화
self.layer4 = torch.nn.Sequential(
    self.fc1,
    torch.nn.ReLU(),
    torch.nn.Dropout(p=1 - self.keep_prob))
# L5 Final FC 625 inputs -> 10 outputs
self.fc2 = torch.nn.Linear(625, 10, bias=True)
torch.nn.init.xavier_uniform_(self.fc2.weight)     # 가중치 초기화
```

**Layer 3 :**

(?, 7, 7, 64)

⋮ Conv(합성곱연산)

너비: $\frac{7-3+2}{1}+1=7$

높이: $\frac{7-3+2}{1}+1=7$

(?, 7, 7, 128)

⋮ Pool(맥스풀링)

너비: $\frac{7-2}{2}+1=3$

높이: $\frac{7-2}{2}+1=3$

(?, 4, 4, 128)

**Fc 1:**

(?, 4, 4, 128)

⋮ 특성맵 펼치기

Inputs 4*4*128

⋮

outputs 625

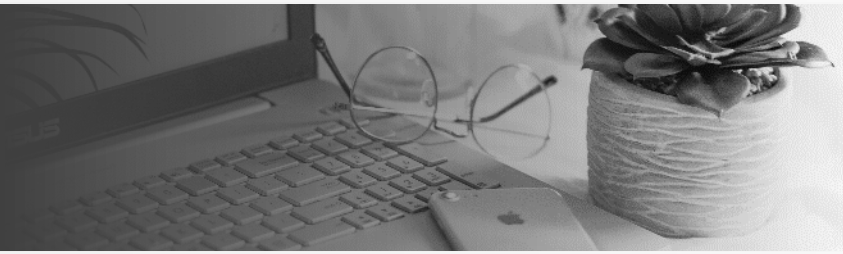
**Fc 3:**

Inputs 625

⋮

Ouputs 10

# 2. 깊은 CNN으로 MNIST 분류하기



```
def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = self.layer3(out)
    out = out.view(out.size(0), -1)   # Flatten them for FC
    out = self.layer4(out)
    out = self.fc2(out)
    return out
```

<- 첫번째 차원인 배치 차원은 그대로 두고 나머지는 펼침

```
# CNN 모델 정의
model = CNN().to(device)
```

# CNN 모델 정의

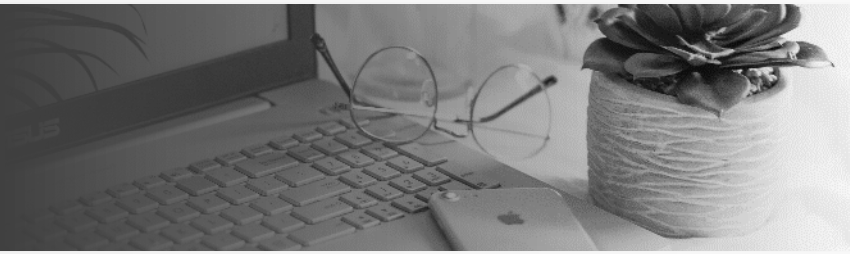
```
criterion = torch.nn.CrossEntropyLoss().to(device)    # 비용 함수에 소프트맥스 함수 포함되어져 있음.
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

# 비용함수, 옵티마이저 정의

```
total_batch = len(data_loader)
```

# 총 배치의 수 구하기 -> 600

# 2. 깊은 CNN으로 MNIST 분류하기

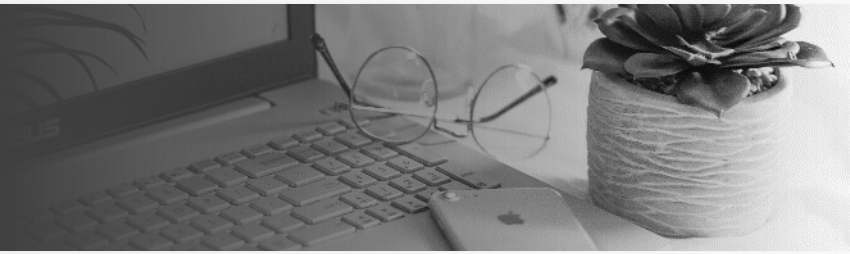


```
for epoch in range(training_epochs):                                    # 모델 훈련
    avg_cost = 0

    for X, Y in data_loader: # 미니 배치 단위로 꺼내온다. X는 미니 배치, Y는 레이블
        # image is already size of (28x28), no reshape
        # label is not one-hot encoded
        X = X.to(device)
        Y = Y.to(device)

        optimizer.zero_grad()
        hypothesis = model(X)
        cost = criterion(hypothesis, Y)
        cost.backward()
        optimizer.step()

        avg_cost += cost / total_batch

    print('[Epoch: {:>4}] cost = {:>.9}'.format(epoch + 1, avg_cost))
```

총 배치의 수 600개, 배치 크기 100
-> 훈련 데이터는 총 60,000개

```
[Epoch:    1] cost = 0.194086716
[Epoch:    2] cost = 0.0506843254
[Epoch:    3] cost = 0.0358644836
[Epoch:    4] cost = 0.0293098353
[Epoch:    5] cost = 0.0240665283
[Epoch:    6] cost = 0.0197937172
[Epoch:    7] cost = 0.0155150732
[Epoch:    8] cost = 0.0159609281
[Epoch:    9] cost = 0.0140717914
[Epoch:   10] cost = 0.0105465455
[Epoch:   11] cost = 0.0104774414
[Epoch:   12] cost = 0.00805889443
[Epoch:   13] cost = 0.00887765829
[Epoch:   14] cost = 0.00816942286
[Epoch:   15] cost = 0.00737793744
```

# 2. 깊은 CNN으로 MNIST 분류하기



```
# 학습을 진행하지 않을 것이므로 torch.no_grad()                    # 테스트 진행
with torch.no_grad():
    X_test = mnist_test.test_data.view(len(mnist_test), 1, 28, 28).float().to(device)
    Y_test = mnist_test.test_labels.to(device)

    prediction = model(X_test)
    correct_prediction = torch.argmax(prediction, 1) == Y_test
    accuracy = correct_prediction.float().mean()
    print('Accuracy:', accuracy.item())
```

3개의 층일 때: `Accuracy:  0.9894999861717224`

5개의 층일 때: `Accuracy: 0.9835000038146973`

-> 층을 깊게 쌓는 것도 중요하지만, 꼭 깊게 쌓는 것이 정확도를 올려주지는 않으며 효율적으로 쌓는 것도 중요